ICASE REPORT

PREPAGING AND APPLICATIONS TO THE STAR-100 COMPUTER

Kishor S. Trivedi

Report Number 76-28

August 30, 1976

INSTITUTE FOR COMPUTER APPLICATIONS

IN SCIENCE AND ENGINEERING

Operated by the

UNIVERSITIES SPACE RESEARCH ASSOCIATION

at

NASA'S LANGLEY RESEARCH CENTER
Hampton, Virginia

PREPAGING AND APPLICATIONS TO THE STAR-100 COMPUTER

Kishor S. Trivedi

Department of Computer Science

Duke University

ABSTRACT

The use of prepaging is described for the CDC STAR-100 system. A feature known as ADVISE is provided by the system for this purpose. A timing analysis of a matrix multiplication routine is carried out to evaluate the effect of prepaging. Finally, a suggestion for a controlled prepaging algorithm is made.

This report was prepared as a result of work performed under NASA Contract No. NAS1-14101 while the author was in residence at ICASE, NASA Langley Research Center, Hampton, VA 23665.

1. Introduction

The Control Data Corporation (CDC) STAR (STring ARray) computer is a high performance vector machine capable of performing up to 100 million operations per second. Although the size of the main memory is limited to either ½ million or 1 million 64-bit words, the use of a virtual memory capability allows a virtual address space size of 4 trillion 64-bit words. The virtual memory is implemented by means of a paging mechanism. The system provides two page sizes, 512 (small) and 65,536 (large) words, respectively.

Because of a very high speed cpu, the system is highly I/O-bound for many problems (Knight, Poole, and Voigt [75]; Lynch [74]). Since the paging device is a disk, the use of the large page size can reduce the I/O time considerably. Since the total number of large page frames is either 8 or 16, the use of a large page dictates a very small degree of multiprogramming. The second technique to improve the performance is to increase the size of the main memory (Denning [68a],[70]). This allows a larger page allotment per program thus reducing the paging I/O. However, due to the small size of the main memory available on the STAR, thrashing can occur even with a very small degree of multiprogramming (Brandwajn [74], Denning [68a], Trivedi [76b],[76c]). Such an I/O limited and memory limited situation forces very low degrees of multiprogramming. Currently, many jobs are constrained to run in a monoprogramming mode. This coupled with a demand paging environment implies no cpu-I/O overlap whatsoever. In such an environment, one possible

way to introduce the cpu-I/O overlap (and hence improve the throughput) is to use prepaging. Prepaging allows the overlap between the execution and I/O of the same job. Therefore, we expect it to improve the throughput, particularly for low degrees of multiprogramming. For further discussion on the usefulness of prepaging see Joseph [70] and Trivedi [74],[76a],[76b], [76d]. Note that the best that we can hope to do with prepaging is either completely mask the I/O-time or completely mask the cpu-time, whichever is less.

The STAR system provides a feature known as ADVISE which allows prepaging. In Section 2, we describe our efforts to exploit this feature. We have developed a subroutine FPRQST which can be called by a FORTRAN programmer to request the operating system either to prepage or free a page. We also illustrate the use of this subroutine in a program carrying out a matrix-multiplication. When we attempted to test our programs on the system, we found that the ADVISE feature does not work in the current version of the operating system. Hopefully, in a future version of the operating system, this feature will be debugged and the experiments of Section 2 may be carried out.

In Section 3, we do a timing analysis of the matrix multiplication routine with and without prepaging and using both the small and the large pages. We expect that the I/O-time can be reduced by an order of magnitude by switching to large pages. It should also be clear that prepaging has a potential of cutting the total execution time in half for a balanced program.

In the current STAR system a global LRU paging algorithm is used (CDC [75], Knight [73]). If prepaging is allowed in a multiprogramming mode, then a program can very rapidly steal pages away from other programs. This may imply an improvement in the performance of that program but the overall

system performance can suffer. Even in a monoprogramming environment (or using a local replacement algorithm) a user can destroy his own performance if uncontrolled prepaging is permitted. In Section 4, we discuss how prepaging can be controlled.

2. Prepaging on STAR

A program can issue an ADVISE message to the operating system of an anticipated need for virtual space in an attempt to avoid a page fault or to advise the system of pages no longer to be used by this program (CDC [76]). The advise can be issued for at most one large page and at most eight consecutive small pages at a time.

Based on this feature, we have developed a routine FPRQST which can be called by a FORTRAN programmer, as follows.

```
CALL FPRQST (OUT, PGSIZE, PGCNT, VBA)
```

where

OUT is a logical variable indicating the direction of transfer, i.e., OUT = .FALSE. if a page is to be fetched and OUT = .TRUE. if a page is to be freed.

PGSIZE is a bit variable such that PGSIZE = B'0' for small pages and B'1' for large pages.

PGCNT is a 4-bit vector giving the number of pages to be transferred.

VBA is a 64-bit vector (or a descriptor) such that the rightmost 48-bits denote the virtual bit address of the first page to be transferred.

The details of the FPRQST routine are given in the Appendix. We now demonstrate the use of the routine in a program that multiplies the matrices A and B storing the result in C. The matrices A, B and C are N by L, L by R, and N by R, respectively. The matrices are assumed to be stored columnwise, as in STAR FORTRAN (CDC [76a]). Let $A = (a_1, a_2, \ldots, a_L)$, $B = (b_1, b_2, \ldots, b_R)$ and $C = (C_1, C_2, \ldots, C_R)$, where a_i denotes the i th column of A. The algorithm we use forms the successive columns of C using the formula,

$$c_j = \sum_{k=1}^{L} b_{kj}\ a_k\ .$$

The corresponding program in STAR FORTRAN can be written as follows.

```
      DO     100     J = 1, R
        DO     10      K = 1, L
          C(1,J;N) = C(1,J;N) + B(K,J)*A(1,K;N)
10      CONTINUE
100   CONTINUE
```

Note that the notation A(i,j;n) denotes a vector consisting of n contiguous array elements with the first element being A(i,j).

During the j th execution of the major loop, the program operates on the j th columns of both B and C and the whole array A. After the j th execution, the j th columns of B and C can be discarded from the main memory and the j+1 st columns of B and C can be brought in. However, the data transfer can only be done in the units of a page size

(assumed to be equal to Z words). Let us assume that $\max(N,L,R) \leq Z$. Let DWS(j) denote the data working set of the program during the j th loop, and let P(x) denote the pages containing the subarray x. Then after a little thought, we arrive at the following expression for DWS.

$$DWS(1) = P(A) \cup P(b_1) \cup P(c_1)$$

$$DWS(j+1) \begin{cases} = DWS(j) & \text{if } \left\lfloor \frac{jL}{Z} \right\rfloor = \left\lfloor \frac{(j-1)L}{Z} \right\rfloor \text{ and } \left\lfloor \frac{jN}{Z} \right\rfloor = \left\lfloor \frac{(j-1)N}{Z} \right\rfloor \\ = DWS(j) - P(b_j) \cup P(b_{j+1}) & \text{if } \left\lfloor \frac{jL}{Z} \right\rfloor > \left\lfloor \frac{(j-1)L}{Z} \right\rfloor \\ = DWS(j) - P(c_j) \cup P(c_{j+1}) & \text{if } \left\lfloor \frac{jN}{Z} \right\rfloor > \left\lfloor \frac{(j-1)N}{Z} \right\rfloor \\ = DWS(j) - P(b_j) - P(c_j) \cup P(b_{j+1}) \cup P(c_{j+1}) & \text{if } \left\lfloor \frac{jL}{Z} \right\rfloor > \left\lfloor \frac{(j-1)L}{Z} \right\rfloor \text{ and } \left\lfloor \frac{jN}{Z} \right\rfloor > \left\lfloor \frac{(j-1)N}{Z} \right\rfloor . \end{cases}$$

The size of the DWS is constant and is equal to $\left\lceil \frac{N*L}{Z} \right\rceil + 2$ pages. The above analysis suggests that we insert the code to prepage the first page of both B and C, and all the pages of A just before the J-loop. Between the statements labelled 10 and 100, we can insert the code to free and prepage one page each of B and C, conditionally. However, this procedure does not allow much overlap between the computation and the page transfer. To allow the desired overlap we assume that the program is allotted 2M (M = 1, 2, ...) page frames more than required to accommodate its DWS. If both N and L divide Z then M = 1 is adequate. Otherwise

a larger value of M is desirable to achieve enough overlap. Finally, to make sure that the three arrays are aligned on a page boundary, each of them should be declared separately in COMMON. With these modifications, the program appears as follows.

```
    prepage the first page of both  B  and  C
    prepage the whole array  A
    prepage the pages numbered  2, ..., M+1  of both  B  and  C
    DO      100      J = 1, R
       DO      10       K = 1, L
          C(1,J;N) = C(1,J;N) + B(K,J)*A(1,K;N)
10     CONTINUE
       IF (J*N/Z .EQ. (J-1)*N/Z)  GO TO 50
       free the page numbered  J*N/Z  of the array  C
       IF ((J*N/Z+M+1) .GT. ⌈N*R/Z⌉)  GO TO 50
       prepage the page numbered (J*N/Z+M+1) of the array  C
50     IF (J*L/Z .EQ. (J-1)*L/Z)  GO TO 100
60     free the page numbered  J*L/Z  of the array  B
       IF ((J*L/Z+M+1) .GT. ⌈L*R/Z⌉)  GO TO 100
       prepage the page numbered (J*L/Z+M+1) of the array  B
100 CONTINUE.
```

We will now show how to code the calls for prepaging and freeing using the FPRQST subroutine. We will only do this for the statement labelled 60; others can be translated similarly. This can be done by replacing the statement 60 by the following statement sequence.

```
    ASSIGN DB, B(1, J; Z)
    CALL FPRQST (.TRUE., PGSIZE, B'0001', DB)
```

Note that DB is assumed to be declared as a descriptor to the array B.

Unfortunately, the ADVISE feature does not work in the current version of the STAR operating system. Therefore, we could not test these programs

$$T_p = \left(\frac{N*L}{Z} + 1\right)T_Z + \left(\frac{R*L}{Z} - 2\right) \max\left\{\frac{ZT_e}{L}, (x+1)T_Z\right\} + 2 \max\left\{\frac{ZT_e}{L}, xT_Z\right\} + T_Z.$$

An expression for T_e is easily derived as follows. During each iteration of the inner loop, one vector multiply (of length N) and one vector addition (of length N) are carried out. The times for these two are given by $159 + N$ and $71 + \frac{N}{2}$ in machine cycles of 40 ns each (CDC [76c]).

$$T_e = ((159 + N)\ L + (71 + \tfrac{1}{2}N)L)*40*10^{-6}\ \text{ms} = (60N*L + .92*10^4 L) * 10^{-6}\ \text{ms}.$$

CDC model 819 disk drive is used as paging device. To simplify our analysis, we assume that the three matrices are stored on separate disk drives, and that each matrix is stored sequentially cyclinder by cylinder. Then the average time to fetch a page can be expressed as

$$T_Z = T_{seek} + T_{latency} + T_{transfer} = N_{seek} * D_{seek} + \frac{1}{2} D_R + \frac{Z}{R}$$

where

D_{seek} is the time to move the disk head between the adjacent cylinders

N_{seek} is the average seek distance

D_R is the disk rotation time

R is the transfer rate in bits per millisecond.

and could not measure the performance improvement obtained by prepaging. Instead, we carry out a timing analysis of the two programs in the next section.

3. The Timing Analysis

Let T_Z denote the average time to fetch (or push) a page. Let T_d and T_p denote the total times to execute the matrix multiplication program of the previous section with demand paging and prepaging, respectively. We will assume that it takes zero time to push a page that is not modified. Similarly, it takes zero time to fetch a page that is uninitialized. Under demand paging, with a page allotment $\geq$ the size of the DWS $\left(= \left\lceil \frac{N*L}{Z} \right\rceil + 2 \right)$, we need to fetch $\left\lceil \frac{N*L}{Z} \right\rceil$ pages of A, $\left\lceil \frac{L*R}{Z} \right\rceil$ pages of B and we need to push $\left\lceil \frac{N*R}{Z} \right\rceil$ pages of C. Let us denote the cpu time spent in a single iteration of the major loop by T_e. Then

$$T_d = \left(\left\lceil \frac{N*L}{Z} \right\rceil + \left\lceil \frac{L*R}{Z} \right\rceil + \left\lceil \frac{N*R}{Z} \right\rceil \right) T_Z + RT_e .$$

Note that T_d is a lower bound which assumes a perfect sequence of replacement as in Belady's MIN algorithm (Belady [66]).

For prepaging, we assume a page allotment = $\left\lceil \frac{N*L}{Z} \right\rceil + 4$. Since the total number of large pageframes available for data is either 7 or 15 in the STAR system, the largest matrices that we can deal with are given by $\left\lceil \frac{N*L}{Z} \right\rceil = 3$ or 12. Since, for a large page, $Z = 65{,}536$, fairly large matrices are considered. To simplify the analysis, we assume that L divides N, and both R and N divide Z. Further let $N = xL$. After a little thought, an upperbound to T_p is easily found to be

For the 819 disk, $D_{seek} = 15$ ms., $D_R = 33$ ms., and $R = 38 * 10^3$ bits per ms. (CDC [74]). N_{seek} is difficult to estimate in general; however, the assumptions made on data layout and the fact the the program is running in a monoprogramming environment make reasonable estimates possible. The capacity of an 819 cylinder is very close to the size of a large page. Therefore, $N_{seek} = 1$ for $Z = 65,536$. Also, $N_{seek} \simeq \frac{1}{129}$ for the small pages.

Therefore, for

$$Z = 65,536 = 2^{16}, \quad T_Z = 15 + 16.5 + \frac{2^{16}*64}{38*10^3} \simeq 140 \text{ ms.},$$

and for

$$Z = 512 = 2^9, \quad T_Z = \frac{15}{129} + 16.5 + \frac{2^9*64}{38*10^3} \simeq 30 \text{ ms.}$$

An an example, let $L = 32$, $N = R = 1024$. Using small pages and demand paging, the total execution time is given by

T_d(small page) $\simeq$ 67,520 ms.

If we introduce prepaging but retain the small page size,

T_p(small page) $\simeq$ 65,280 ms.

In this case, we see that a small percentage reduction in the total time is obtained due to prepaging. The reason for this behaviour is that the program is very highly I/O bound. In fact the total I/O time is 65,280 ms., and

the total computation time is 2240 ms. The best that can be done by prepaging is to mask all the computation time.

Now for the same problem size, let us use a large page. In this case,

T_d(large page) = 4760 ms.

We see that more than an order of magnitude reduction in the total time is obtained by switching to a large page. Also,

T_p(large page) = 2660 ms.

Thus, keeping the large page size and using prepaging further obtains a 45% reduction in the total execution time. In this case, we see that the total I/O time is 2520 ms. and the total computation time is 2240 ms. The best that can be done by prepaging is to mask all the computation time. Our prepaging scheme performs very close to the ideal prepaging scheme in this case. It is easy to see that for a nearly balanced program close to 50% reduction in the total execution time can be accrued due to prepaging. However, for a highly imbalanced program, relatively small percentage improvement can be expected. If the program is heavily I/O bound then the use of the large page size can improve the performance by an order of magnitude.

The following table gives the total execution time for different problem sizes with and without prepaging (assuming Z = 65,536).

N	L	R	T_d ms.	T_p ms.
1024	32	1024	4760	2660
1024	64	1024	7000	4900
512	512	512	11920	11080
512	32	512	3160	2520

4. Controlled Prepaging

In the current version of the STAR operating system, a global LRU algorithm is used for the page replacement decision. Furthermore, an advise to bring in a page is allowed to increase the page allotment of a program at the expense of some other program. In such an environment, a programmer interested in optimizing his own performance can quickly force out all the other programs from the main memory. With this type of environment, prepaging can only be recommended in a monoprogramming application. Even in a monoprogramming environment (or in multiprogramming environment with a local replacement algorithm) a naive user can fill up valuable space with almost useless pages thus destroying his own performance. The paging algorithm should guard against such possibilities as far as possible.

The first requirement of controlled prepaging is, then, that the user spaces be isolated from each other so as to implement a local paging algorithm in each individual space. Such a situation is desirable even in a demand paging environment (Denning [68b],[70]; Knight [73]). However, if we require a total isolation of the user spaces from each other then we are constrained to the use of a fixed-memory paging algorithm. Since a program's memory requirements are expected to change during its execution, a variable-memory paging algorithm can potentially perform better than a fixed-memory algorithm. To resolve this issue, we allow the user space allocation to vary dynamically, however, the variation is not controlled by the user but is controlled by the system. The space allocation may be based on the performance of a program in the last interval of measurement. For further details see Chu and Opderbeck [72], Denning [72], Denning and Schwarz [72] and Knight [73]. With this approach, over short time intervals the space

allocation can be assumed to be constant and a fixed-memory paging algorithm can be utilized.

In order to prevent a user from misusing the memory space allotted to him we postulate the following requirements. When a user expresses the desire to prepage a page, this is not taken as a command but as an advice to the operating system. The operating system may execute it immediately, may defer it or even ignore it. Thus not only a user cannot steal a page away from another user by issuing a prepage request, but he cannot steal a page away from his own space unless the operating system permits it. The system does not allow a prepaged page to be brought in if this requires the replacement of a useful page. Similarly, when a program issues a request to free a page the empty space thus created remains in the space allotment of the program. Thus the space for prepaging is provided by the programmer himself. We assume that when a programmer issues a prepage request he is reasonably certain of the use of that page. Therefore, to avoid an extra page fetch, we require that a prepaged page cannot be replaced until after its first use. The procedure outlined so far is likely to deadlock as follows. Initially when all the space allocated to a program is empty, a programmer can issue repeated prepaging requests filling up the entire space. If, before using any of these pages, he requires (or page faults) another page then a deadlock will occur. To avoid the deadlock, we allow the prepaged pages to fill up at most a fraction $0 \leq b < 1$ of the allocated space.

Let N denote the program's address space and let S_t denote the memory state at time t. Let c be the page allotment. The memory state S_t is divided into four disjoint sets: D_t denotes that set of pages which have been declared dead (or free), N_t denotes the set of pages which have been prepaged but not yet set up in the main memory, P_t denotes

the set of pages which have been prepaged and set up in memory but not yet used, and $U_t = S_t - P_t - D_t - N_t$ denotes the rest of the pages which have been used at least once. Let $R_A(S,q,x)$ denote the page replaced by the paging algorithm A given that the memory state is S, the control state is q and the page x is to be brought into the main memory. The control state q imposes an ordering on the memory state S to help make the replacement decision. We assume that the advise given by the programmer is either PRE(x) or FREE(x) for some $x \epsilon N$. If we denote the reference string of the program by $r_1, \ldots r_t, \ldots$ then $r_t \in N$ or $r_t = PRE(x)$ for $x \in N$ or $r_t = FREE(x)$ for $x \in N$. Based on a demand paging algorithm A, we now define a prepaging algorithm FPA.

FPA:

[Step 1] IF $r_{t+1} = x \in N$ THEN

[a] IF $x \in N_t$ THEN

/* THIS IS ALMOST LIKE A PAGE FAULT.
THE PROGRAM HAS TO BE SUSPENDED
UNTIL THE REQUIRED PAGE IS SET-UP
IN THE MAIN MEMORY, AFTER WHICH */

$U_{t+1} = U_t + x$;

$N_{t+1} = N_t - x$ and RETURN;

[b] IF $x \in P_t$ THEN /* SUCCESS */

$P_{t+1} = P_t - x$;

$U_{t+1} = U_x + x$ and RETURN;

[c] IF $x \in U_t$ THEN /* SUCCESS */

$U_{t+1} = U_t$ and RETURN;

[d] IF $x \notin S_t$ THEN /* PAGE FAULT */

IF $|S_t| < c$ THEN

$U_{t+1} = U_t + x$;

ELSE

$U_{t+1} = U_t + x - R_{FA}(U_t \cup D_t, q, x)$ and RETURN;

[Step 2] IF $r_{t+1} = FREE(x)$ THEN

IF $x \in U_t$ THEN

$U_{t+1} = U_t - x$; $D_{t+1} = D_t + x$; and RETURN;

IF $x \in P_t$ THEN

$P_{t+1} = P_t - x$; $D_{t+1} = D_t + x$; and RETURN;

IF $x \in N_t$ THEN

$N_{t+1} = N_t - x$; $D_{t+1} = D_t + x$; and RETURN;

[Step 3] IF $r_{t+1} = PRE(x)$ THEN

IF $|S_t - D_t| < c$ and $|P_t \cup N_t| < b * c$ and $x \notin S_t$

THEN

$N_{t+1} = N_t + x$;

/* INSTRUCTIONS TO FETCH THE PAGE ARE

ISSUED NOW. LATER, WHEN THE PAGE IS SET UP

IN THE MAIN MEMORY IT WILL BE INTRODUCED IN

THE SET P_t AND TAKEN OFF FROM THE SET N_t

BY SOME OTHER MODULE OF THE OPERATING SYSTEM */

IF $|S_t| = c$ THEN

$D_{t+1} = D_t - y$ FOR SOME $y \in D_t$;

END FPA

where

$$R_{FA}(U_t \cup D_t, q, x) \begin{cases} = y \text{ for some } y \in D_t, \text{ if } D_t \neq \phi \\ = R_A(U_t, q, x) \text{ otherwise.} \end{cases}$$

5. Conclusion

Many programs operating on large arrays are I/O-bound in the STAR system, and because of the relatively small memory, are forced to run in a monoprogramming environment. It is shown that the use of large pages for these arrays can reduce the I/O time by more than an order of magnitude and thus make it more balanced. Monoprogramming and demand paging imply that no cpu-I/O overlap is achieved. The use of prepaging introduces the overlap and thus reduces the total execution time of the program. If the program is balanced then a substantial reduction in the total execution time is shown to be achieved by prepaging.

The STAR system provides a feature known as ADVISE which supports prepaging. However, this feature does not work at present. The results of this paper will, hopefully, serve as an impetus to debug the feature. Even when this feature starts functioning, we can only recommend prepaging for a monoprogramming environment. We also provide a suggestion for a controlled prepaging algorithm which can be usefully implemented in a multiprogramming environment.

6. Acknowledgement

I would like to thank John Knight and Robert Voigt for helpful suggestions during the course of this work.

REFERENCES

1. Belady [66]. Belady, L. A.: "A Study of Replacement Algorithms for a Virtual Storage Computer." IBM Systems Journal 5, 2 (1966), pp. 78-101.

2. Brandwajn [74]. Brandwajn, A.: "A Model of a Time Sharing Virtual Memory System Solved Using Equivalence and Decomposition Methods." Acta Informatica 4, pp. 11-47.

3. CDC [74]. Control Data Corporation STAR-100 Peripheral Stations, Revision B, Arden Hills, Minn.

4. CDC [75]. Control Data Corporation STAR-100 Hardware Reference Manual, St. Paul, Minn.

5. CDC [76]. Control Data Corporation STAR-100 Operating System Reference Manual, Sunnyvale, California.

6. CDC [76a]. Control Data Corporation STAR-100 FORTRAN Language Reference Manual, Sunnyvale, California.

7. CDC [76b]. Control Data Corporation STAR-100 Assembler Reference Manual, Sunnyvale, California.

8. CDC [76c]. Control Data Corporation STAR-100 Preliminary Instruction Execution Timing Manual, Arden Hills, MN.

9. Chu and Opderbeck [72]. Chu, W. W. and Opderbeck, H.: "The Page Fault Frequency Replacement Algorithm," Proc. AFIPS FJCC, 1972, pp. 597-609.

10. Denning [68a]. Denning, P. J.: "Thrashing: Its Causes and Prevention," Proc. AFIPS SJCC 33 (1968), pp. 915-922.

11. Denning [68b]. Denning, P. J.: "The Working Set Model of Program Behaviour," CACM 11, 5 (1968), pp. 323-333.

12. Denning [70]. Denning, P. J.: "Virtual Memory," Computing Surveys 2, 3 (Sept. 1970), pp. 153-189.

13. Denning [72]. Denning, P. J.: "On Modelling Program Behavior," Proc. AFIPS SJCC, 1972.

14. Denning and Schwartz [72]. Denning, P. J. and Schwartz, S. C.: "Properties of the Working Set Model," CACM 15, 3 (Mar. 1972), pp. 191-198.

15. Joseph [70]. Joseph, M.: "An Analysis of Paging and Program Behaviour," The Computer Journal 13, 1 (Feb. 1970), pp. 48-54.

16. Knight [73]. Knight, J. C.: "Scheduling Central Resources on the CDC STAR 100," ICASE Report, NASA Langley Research Center, Hampton, Virginia, August 1973.

17. Knight,Poole,and Voigt [75]. Knight, J. C.; Poole, W. G.; and Voigt, R. G.: "System Balance Analysis for Vector Computers," Proc. ACM Annual Conference, October 1975, pp. 163-168.

18. Lynch [74]. Lynch, W. C.: "How to Stuff an Array Processor," Third Texas Conference on Computing Systems, Nov. 1974.

19. Trivedi [74]. Trivedi, K. S.: "Prepaging and Applications to Structured Array Problems," Ph.D. Thesis, University of Illinois, Urbana, June 1974.

20. Trivedi [76a]. Trivedi, K. S.: "Prepaging and Applications to Array Algorithms," to appear in IEEE Transactions on Computers.

21. Trivedi [76b]. Trivedi, K. S.: "On Some Aspects of the Performance of Paging Systems," submitted for publication, January 1976.

22. Trivedi [76c]. Trivedi, K. S.: "A Performance-Comparison of Systems With and Without Virtual Memory," submitted for publication, June 1976.

23. Trivedi [76d]. Trivedi, K. S.: "On the Paging Performance of Array Algorithms," submitted for publication, May 1976.

APPENDIX

Before issuing an ADVISE to the operating system, a three word message called the alpha packet has to be formed. The format of the message is as follows (CDC [76]):

Alpha (1)	r (16)	len (16)	unused (16)	0007 (16)
Alpha (2)	unused (16)	eea (48)		
Alpha (3)	ss (8)	pgct (8)	vba (48)	

r — Response code returned by operating system when this message has been processed. If no error occurs, the response code is zero.

len — If len = FFFF, Alpha (3) contains the length and virtual bit address of the Beta portion of the message. Otherwise, Beta is assumed to begin at Alpha (3), and len is the length of the Beta portion.

eea — Virtual bit address to receive control if an error occurs during message processing ($r \neq 0$). If eea = 0 the error is considered fatal.

ss — Error response field:

pgct Page control fields further divided as follows:

pio

0 = Attach or load pages

1 = Remove pages

psz

0 = Small pages

1 = Large page

pn Page count with a maximum value of eight for small pages.

vba Virtual bit address referred to by the FPRQST action.

After forming the message, the user issues an exit force instruction that transfers the control to the operating system. Immediately following the exit force instruction in the instruction stream is a 32-bit indirect message pointer: Hexadecimal format of the message pointer is:

00EE00rr

where rr is the register containing the virtual address of the message.

The following FORTRAN subroutine FPRQST forms the message in the 192-bit vector MSG based on the actual parameters: OUT, PGSIZE, PGCNT and VBA. The error exit address (eea) is taken to be the address of the statement labelled 50. After forming the message, FPRQST calls an assembly language routine PREPAG which issues the exit force and also plants the indirect message pointer as required. For details of the STAR assembly language see CDC [76b]. A descriptor R, pointing to the message is passed as a parameter to the PREPAG routine.

```
      SUBROUTINE FPRQST  (OUT,PGSIZE,PGCNT,VBA)
         INTEGER CB,CI,L50
         BIT MSG(192)
         EQUIVALENCE (L50,MSG(65))
         LOGICAL OUT
         BIT  PGCNT(4),VBA(64),PGSIZE,E,R
         DESCRIPTOR E,R
         ASSIGN R,MSG(1;192)
         R=B'00000000000000000000000000001000000000000000000
     X       000000000000111'
C        THE FOLLOWING FIVE STATEMENT SEQUENCE FORMS THE VIRTUAL
C        ADDRESS OF STATEMENT 50 IN THE INTEGER L50,SINCE L50 IS
C        EQUIVALENCED TO THE SECOND WORD OF MSG, THE EEA FIELD GETS THE
C        CORRECT ERROR EXIT ADDRESS,FOR DETAILS SEE STAR FORTRAN +
C        META MANUALS
         CALL Q8BSAVE(3, ,3)
 5       CALL Q8EX    (4,&5)
         CALL Q8SUBX  (3,4,CB)
         CALL Q8EX    (CI,&50)
         L50=CI+CB
         ASSIGN E,MSG(138;1)
         E=PGSIZE
         ASSIGN E,MSG(137;1)
         E=B'0'
         IF (OUT)  E= B'1'
         ASSIGN E,MSG(141;4)
         E=PGCNT(1;4)
         ASSIGN E,MSG(145;48)
         E=VBA(17;48)
         CALL PREPAG(R)
         GOTO 5000
 50      PRINT 500
 500     FORMAT(5X,'YOUR ADVISE HAS BOMBED')
 5000    RETURN
      END


         OUTPUT
         LIBP,STDLIB
PREPAG   IDENT
         INITIAL
         REG,128    R,DUM
PREPAG   BEGIN
         PARAMS,ALL,2    R
         LOD        R,DUM
         OSCALL     DUM
         RETURN
         FINAL
         END
         FINIS
```